链滴

设计模式 |05 抽象工厂模式

作者：qq692310342
原文链接：https://ld246.com/article/1567649176125
来源网站：链滴

开头说几句

博主的博客地址： https://www.jeffcc.top/

博主学习设计模式用的书是Head First的《设计模式》，强烈推荐配套使用！

什么是抽象工厂模式

权威定义：抽象工厂模式提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体的。

博主的理解：抽象工厂的任务就是定义一个负责创建一组产品的接口，接口的内部的每个方法都负责建一个具体的产品，所以可以理解为抽象工厂就是多个工厂方法的高聚合的体现。

抽象工厂模式和工厂方法模式的比较

1. 工厂方法使用继承，把对象的创建委托给子类，子类实现工厂方法来创建对象；
2. 抽象工厂使用的是对象组合，对象的创建被实现在工厂接口所暴露出来的方法中；
3. 工厂方法允许类将实例化延迟到子类进行；
4. 抽象工厂创建相关的对象家族，而不需要依赖他们的具体类；

设计原则

1. 依赖倒置原则，要依赖抽象，不要依赖具体类，具体做法是需要高层组件（工厂）和底层组件（实类）之间不要有太多的依赖关系，而是可以通过一个共同的抽象类（工厂产生的对象）来实现依赖倒。
2. 多用组合，少用继承。
3. 针对接口编程，而不是针对实现编程。

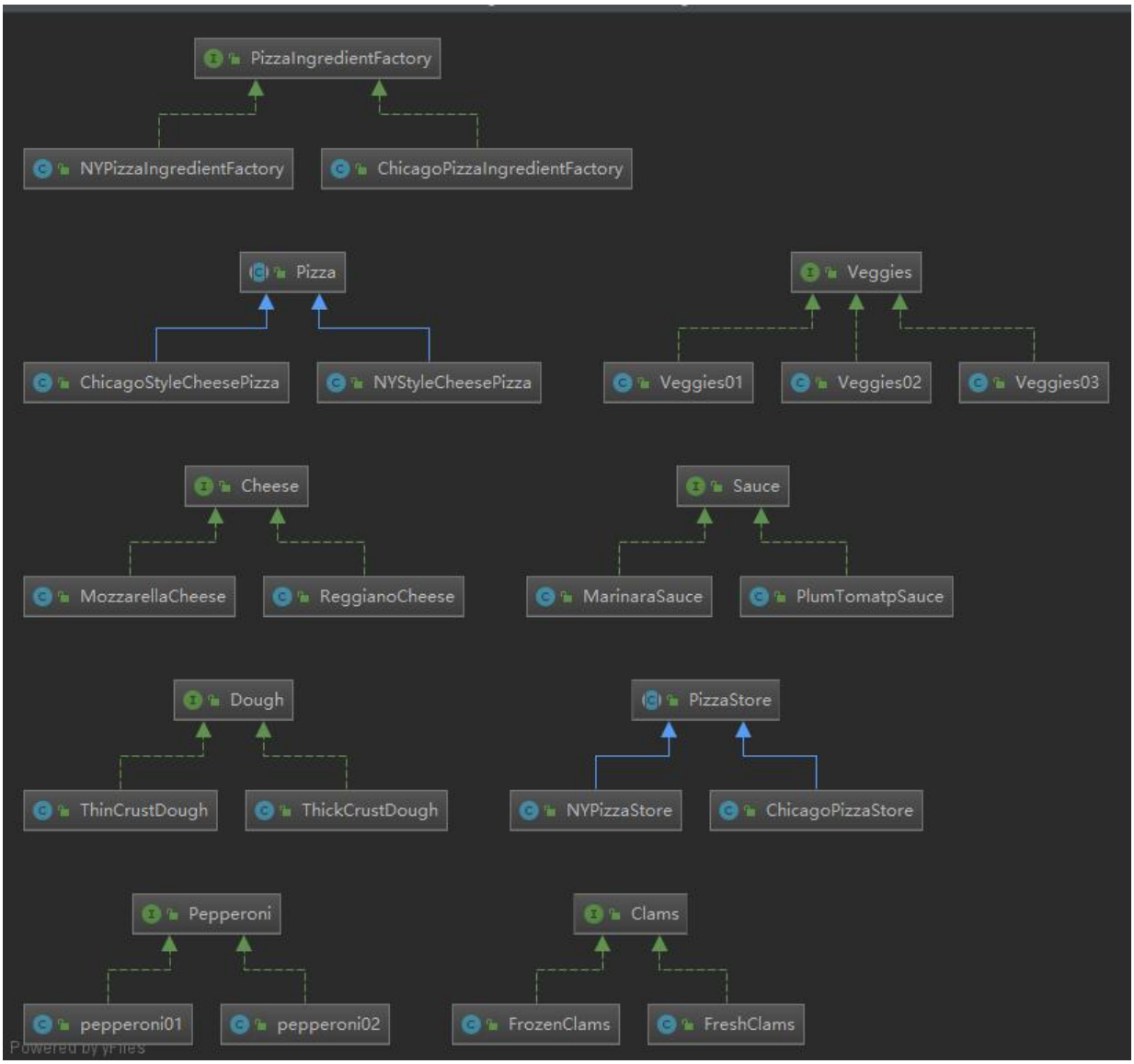
4. 为交互对象之间的松耦合设计而努力。
5. 类应该对外开放拓展，对修改关闭。

设计实例

设计背景

利用上一篇博客讲到的pizza店为拓展，为了规范化加盟店，新加一个原料工厂，各个地区有自己不的风格的原料，但是都必须遵守原料工厂所提供的的原料标准（接口规范化）

项目类图



项目结构

披萨工厂

```
package factory;

import pizza.Pizza;

/**
 * 披萨工厂
 */
public abstract class PizzaStore {
```

```
    public Pizza orderPizza(String type){
        Pizza pizza;
//        进行创建pizza
        pizza = createPizza(type);
//        下面进行的是共同的东西，所以封装起来到工厂中
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;

    }

    /**
     * 抽取出一个抽象的制作pizza的方法来交给子类具体实现
     * 工厂模式的关键所在
     * @param type  什么风味的pizza
     * @return
     */
    public abstract Pizza createPizza(String type);

}
```

ChicagoPizzaStore加盟店

```
package store;

import factory.PizzaStore;
import ingredientfactory.PizzaIngredientFactory;
import ingredientfactory.impl.ChicagoPizzaIngredientFactory;
import pizza.Pizza;
import pizza.impl.ChicagoStyleCheesePizza;

/**
 * ChicagoPizzaStore加盟店
 */
public class ChicagoPizzaStore extends PizzaStore {

//    配置芝加哥原料工厂
    PizzaIngredientFactory ingredientFactory = new ChicagoPizzaIngredientFactory();
    /**
     * 模拟测试生产pizza  太多类型了就写一个 理论可以无限拓展
     * @param type 什么风味的pizza
     * @return
     */
    @Override
    public Pizza createPizza(String type) {
        if ("cheese".equals(type)) {
            return new ChicagoStyleCheesePizza(ingredientFactory);
        } else {
            return null;
        }
```

```
    }
}
```

纽约pizza加盟店

```
package store;

import factory.PizzaStore;
import ingredientfactory.PizzaIngredientFactory;
import ingredientfactory.impl.NYPizzaIngredientFactory;
import pizza.Pizza;
import pizza.impl.NYStyleCheesePizza;

/**
 * 纽约pizza加盟店
 */
public class NYPizzaStore extends PizzaStore {

//    配置原料加工工厂
    PizzaIngredientFactory ingredientFactory = new NYPizzaIngredientFactory();
    /**
     * 模拟测试生产pizza  太多类型了就写一个 理论可以无限拓展
     * @param type 什么风味的pizza
     * @return
     */
    @Override
    public Pizza createPizza(String type) {
        if ("cheese".equals(type)) {
            return new NYStyleCheesePizza(ingredientFactory);
        } else {
            return null;
        }
    }
}
```

Pizza抽象类（关键）

```
package pizza;

import ingredient.*;

/**
 * pizza的抽象类
 * 也就是之前说的依赖倒置所依赖的类!!!
 * 需要有pizza的操作的方法
 */
public abstract class Pizza {

    public String name;
    public Cheese cheese;
    public Clams clams;
```

```
    public Dough dough;
    public Pepperoni pepperoni;
    public Sauce sauce;
    public Veggies[] veggies;

    /**
     * 提供默认的基本做法 进行烘烤、切片、装盒 如果需要可以进行重写
     */
//    对准备方法进行抽象化  接入工厂的原料准备！
    public abstract void prepare();

    /**
     * 烘烤
     */
    public void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    /**
     * 切片
     */
    public void cut() {
        System.out.println("Cutting the pizza into diagonal slices!");
    }

    /**
     * 装盒
     */
    public void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName() {
        return name;
    }
}
```

ChicagoStyleCheesePizza 芝加哥风味

```
package pizza.impl;

import ingredient.Veggies;
import ingredientfactory.PizzaIngredientFactory;
import pizza.Pizza;

/**
 * 实现不同的风味的pizza ChicagoStyleCheesePizza
 */
public class ChicagoStyleCheesePizza extends Pizza {

    PizzaIngredientFactory pizzaIngredientFactory;

//    初始化pizza风味需要的原料工厂
```

```
    public ChicagoStyleCheesePizza(PizzaIngredientFactory pizzaIngredientFactory) {
        this.pizzaIngredientFactory = pizzaIngredientFactory;
        super.name = "ChicagoStyleCheesePizza";
    }

    /**
     * 设置风味
     * 变化之处!
     * 原料从工厂中获得!
     */
    @Override
    public void prepare() {
        System.out.println("Preparing:"+super.getName());
        System.out.println("Using Ingredient:");
//        所有的原料都是从工厂中获得的 而且是成套获得 这体现了抽象工厂方法的优势 将所有的工厂合到一个接口中
        cheese = pizzaIngredientFactory.createCheese();
        clams = pizzaIngredientFactory.createClams();
        dough = pizzaIngredientFactory.createDough();
        pepperoni = pizzaIngredientFactory.createPepperoni();
        sauce = pizzaIngredientFactory.createSauce();
        veggies = pizzaIngredientFactory.createVeggies();
//        获取到之后进行输出
        System.out.println(cheese.getCheese());
        System.out.println(clams.getClams());
        System.out.println(dough.getDough());
        System.out.println(pepperoni.getPepperoni());
        System.out.println(sauce.getSauce());
        for (Veggies veggy : veggies) {
            System.out.print(veggy.getVeggies()+" ");
        }
        System.out.println();
    }

    /**
     * 模拟需要有不同的切片方法
     */
    @Override
    public void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

纽约风味pizza

```
package pizza.impl;

import ingredient.Veggies;
import ingredientfactory.PizzaIngredientFactory;
import pizza.Pizza;

/**
 * 实现不同的风味的pizza 纽约风味
```

```
 */
public class NYStyleCheesePizza extends Pizza {

    /**
     * 设置风味
     */
    PizzaIngredientFactory pizzaIngredientFactory;

    //    初始化pizza风味需要的原料工厂
    public NYStyleCheesePizza(PizzaIngredientFactory pizzaIngredientFactory) {
        this.pizzaIngredientFactory = pizzaIngredientFactory;
        super.name = "NYStyleCheesePizza";
    }

    /**
     * 设置风味
     * 变化之处!
     * 原料从工厂中获得!
     */
    @Override
    public void prepare() {
        System.out.println("Preparing:"+super.getName());
        System.out.println("Using Ingredient:");
//        所有的原料都是从工厂中获得的 而且是成套获得 这体现了抽象工厂方法的优势 将所有的工厂
合到一个接口中
        cheese = pizzaIngredientFactory.createCheese();
        clams = pizzaIngredientFactory.createClams();
        dough = pizzaIngredientFactory.createDough();
        pepperoni = pizzaIngredientFactory.createPepperoni();
        sauce = pizzaIngredientFactory.createSauce();
        veggies = pizzaIngredientFactory.createVeggies();
        //        获取到之后进行输出
        System.out.println(cheese.getCheese());
        System.out.println(clams.getClams());
        System.out.println(dough.getDough());
        System.out.println(pepperoni.getPepperoni());
        System.out.println(sauce.getSauce());
        for (Veggies veggy : veggies) {
            System.out.print(veggy.getVeggies()+" ");
        }
        System.out.println();
    }
}
```

原料工厂（关键）

```
package ingredientfactory;

import ingredient.*;

/**
 * 原料工厂
 * 提供一套的方法进行原料的创建
```

```
 * 这里体现出了抽象工厂模式的特点
 * 使用组合的方法将多个工厂的方法组合在一起
 */
public interface PizzaIngredientFactory {
    Dough createDough();
    Sauce createSauce();
    Cheese createCheese();
    Veggies[] createVeggies();
    Pepperoni createPepperoni();
    Clams createClams();
}
```

芝加哥的原料提供厂

```
package ingredientfactory.impl;

import ingredient.*;
import ingredient.cheeseImpl.ReggianoCheese;
import ingredient.clamsImpl.FrozenClams;
import ingredient.doughImpl.ThickCrustDough;
import ingredient.pepperoniImpl.pepperoni02;
import ingredient.sauceImpl.PlumTomatpSauce;
import ingredient.veggiesImpl.Veggies02;
import ingredient.veggiesImpl.Veggies03;
import ingredientfactory.PizzaIngredientFactory;

/**
 * 实现了芝加哥的原料提供厂
 */
public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory {
    @Override
    public Dough createDough() {
        return new ThickCrustDough();
    }

    @Override
    public Sauce createSauce() {
        return new PlumTomatpSauce();
    }

    @Override
    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    @Override
    public Veggies[] createVeggies() {
        Veggies[] veggies = {new Veggies02(),new Veggies03()};
        return veggies;
    }

    @Override
    public Pepperoni createPepperoni() {
```

```
        return new pepperoni02();
    }

    @Override
    public Clams createClams() {
        return new FrozenClams();
    }
}
```

纽约的原料提供厂

```
package ingredientfactory.impl;

import ingredient.*;
import ingredient.cheeseImpl.MozzarellaCheese;
import ingredient.clamsImpl.FreshClams;
import ingredient.doughImpl.ThinCrustDough;
import ingredient.pepperoniImpl.pepperoni01;
import ingredient.sauceImpl.MarinaraSauce;
import ingredient.veggiesImpl.Veggies01;
import ingredient.veggiesImpl.Veggies03;
import ingredientfactory.PizzaIngredientFactory;

/**
 * 实现了纽约的原料提供厂
 */
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {
    @Override
    public Dough createDough() {
        return new ThinCrustDough();
    }

    @Override
    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    @Override
    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    @Override
    public Veggies[] createVeggies() {
        Veggies[] veggies = {new Veggies01(),new Veggies03()};
        return veggies;
    }

    @Override
    public Pepperoni createPepperoni() {
        return new pepperoni01();
    }
```

```
    @Override
    public Clams createClams() {
        return new FreshClams();
    }
}
```

原料接口规范

```
package ingredient;

public interface Dough {
    String getDough();
}
```

```
package ingredient;

public interface Cheese {
    String getCheese();
}
```

```
package ingredient;

public interface Clams {
    String getClams();
}
```

```
package ingredient;

public interface Pepperoni {
    String getPepperoni();
}
```

```
package ingredient;

public interface Sauce {
    String getSauce();
}
```

```
package ingredient;

public interface Veggies {
    String getVeggies();
}
```

不同地区的原料实现类

```java
package ingredient.cheeseImpl;

import ingredient.Cheese;

/**
 * cheese原料1 MozzarellaCheese
 */
public class MozzarellaCheese implements Cheese {
    private String cheese = "MozzarellaCheese";

    public String getCheese() {
        return cheese;
    }

}
```

```java
package ingredient.cheeseImpl;

import ingredient.Cheese;

/**
 * cheese原料1 ReggianoCheese
 */
public class ReggianoCheese implements Cheese {
    private String cheese = "ReggianoCheese";

    public String getCheese() {
        return cheese;
    }

}
```

```java
package ingredient.doughImpl;

import ingredient.Dough;

/**
 * 原料2 ThickCrustDough
 */
public class ThickCrustDough implements Dough {
    private String dough = "ThickCrustDough";

    public String getDough() {
        return dough;
    }
}
```

```java
package ingredient.doughImpl;

import ingredient.Dough;
```

```java
/**
 * 原料2 ThinCrustDough
 */
public class ThinCrustDough implements Dough {
    private String dough = "ThinCrustDough";

    public String getDough() {
        return dough;
    }
}
```

```java
package ingredient.clamsImpl;

import ingredient.Clams;

/**
 * 原料3 FreshClams
 */
public class FreshClams implements Clams {
    private String clams = "FreshClams";

    public String getClams() {
        return clams;
    }
}
```

```java
package ingredient.clamsImpl;

import ingredient.Clams;

/**
 * 原料3 FrozenClams
 */
public class FrozenClams implements Clams {
    private String clams = "FrozenClams";

    public String getClams() {
        return clams;
    }
}
```

```java
package ingredient.sauceImpl;

import ingredient.Sauce;

/**
 * 原料4 MarinaraSauce
 */
public class MarinaraSauce implements Sauce {
    private String sauce = "MarinaraSauce";
```

```
    public String getSauce() {
        return sauce;
    }
}


package ingredient.sauceImpl;

import ingredient.Sauce;

/**
 * 原料4 PlumTomatpSauce
 */
public class PlumTomatpSauce implements Sauce {
    private String sauce = "PlumTomatpSauce";

    public String getSauce() {
        return sauce;
    }
}


package ingredient.veggiesImpl;

import ingredient.Veggies;

/**
 * 原料5 Veggies01
 */
public class Veggies01 implements Veggies {
    private String veggies = "Veggies01";

    public String getVeggies() {
        return veggies;
    }
}


package ingredient.veggiesImpl;

import ingredient.Veggies;

/**
 * 原料5 Veggies02
 */
public class Veggies02 implements Veggies {
    private String veggies = "Veggies02";

    public String getVeggies() {
        return veggies;
    }
}


package ingredient.veggiesImpl;
```

```java
import ingredient.Veggies;

/**
 * 原料5 Veggies03
 */
public class Veggies03 implements Veggies {
    private String veggies = "Veggies03";

    public String getVeggies() {
        return veggies;
    }
}
```

```java
package ingredient.pepperoniImpl;

import ingredient.Pepperoni;

/**
 * 原料6 pepperoni01
 */
public class pepperoni01 implements Pepperoni {
    private String pepperoni = "pepperoni01";

    public String getPepperoni() {
        return pepperoni;
    }
}
```

```java
package ingredient.pepperoniImpl;

import ingredient.Pepperoni;

/**
 * 原料6 pepperoni02
 */
public class pepperoni02 implements Pepperoni {
    private String pepperoni = "pepperoni02";

    public String getPepperoni() {
        return pepperoni;
    }
}
```

输出结果

```
PreparingNYStyleCheesePizza
Using Ingredient:
MozzarellaCheese
FreshClams
ThinCrustDough
```

```
pepperoni01
MarinaraSauce
Veggies01 Veggies03
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices!
Place pizza in official PizzaStore box
jay order a NYStyleCheesePizza

PreparingChicagoStyleCheesePizza
Using Ingredient:
ReggianoCheese
FrozenClams
ThickCrustDough
pepperoni02
PlumTomatpSauce
Veggies02 Veggies03
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
jeff order a NYStyleCheesePizza
```

回归定义

我们之前定义说的抽象工厂模式提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定体的类。

在这个具体的实例中都有着对应的关系。

PizzaIngredientFactory提供了一个接口，里面有着一个原料家族的创建的关系，并且这个PizzaIngrdientFactory类并没有指定任何的具体的类；

而我们也很容易看出，如果把原料分离开来单独写，那么就是一个一个的小工厂，所以说抽象工厂模其实通俗来说就是一个大工厂里面有好多个小工厂！

END

2019年9月5日10:05:54